Examining ASP.NET's Membership, Roles, and Profile - Part 12

By Scott Mitchell

Introduction

Several of the earlier installments in this article series examined how to apply authorization rules in order to prohibit particular users, roles, or classes of users from accessing particular resources. For instance, Part 2 showed how to define URL-based authorization rules in `web.config` for roles. With just a bit of XML markup, it is possible to block particular users or roles from visiting certain web pages. Just installments also looked at using the LoginView control, which displays different markup based on whether the user is authenticated or not (and can also be used to display different markup based on the currently logged in user's role). There are also programmatic techniques you can use to determine the identity of the currently logged on user and what roles she belongs to.

The URL-based authorization, LoginView control, and programmatic techniques can be used in tandem to ensure that a user does not visit a page or perform some operation if she is not authorized. But what if you forget to implement one of these safeguards? For example, imagine that you have a web page that includes a button that, when clicked, perform some task that is only intended for administrators. You could put this button in a LoginView control or you could use programmatic techniques to ensure that only users in the appropriate role (say, `Admin`) saw the button. But what if sometime later you, or another developer, removed this check by accident? The net result would be that any user visiting the page could perform the administrator-only operation! Whoops!

To reduce the likelihood of such security mishaps, the .NET Framework includes capabilities for declaratively asserting permissions (via attributes) on methods and classes. In a nutshell, you can add such attributes to ASP.NET pages, their code-behind classes, and your business logic and data access layers. With these attributes in place, your visitors will be barred from performing unauthorized actions, regardless of whether there are any security holes in the user interface. Read on to learn more!

Using the `PrincipalPermissionAttribute` Class to Declaratively Define Authorization Rules

The .NET Framework includes a class named `PrincipalPermissionAttribute` that enables developers to apply principal permissions to code using declarative syntax. This attribute can decorate either classes or methods and its properties spell out what users or roles can access the class or method it decorates. The `PrincipalPermissionAttribute` class has the following important properties:

- `Action` - defines the authorization check that is to be performed. There are a number of different `Action` options. The one you'll use most often is `Demand`, which requires that all callers in the call stack have the permission specified by the other properties.
- `Authenticated` - a Boolean value that indicates whether the caller must be authenticated or not.
- `Name` - the name of the caller. An ASP.NET web application this is typically the username of the currently logged in user.
- `Role` - the name of the role the caller must belong to.

The `Action` property is mandatory, but the other ones can be used in various combinations. For example, to declare that a particular method can only be accessed by users in a particular role, you would use the following syntax:

```
// C#
[PrincipalPermission(SecurityAction.Demand, Role="roleName")]
public void SomeMethod(...)
{
   ...
}


' VB
<PrincipalPermission(SecurityAction.Demand, Role:="roleName")> _
Public Sub SomeMethod(...)
   ...
End Sub
```

Note: the `PrincipalPermissionAttribute` class is located in the `System.Security.Permissions` namespace. You may need to import this namespace into the class files or ASP.net pages that use the `PrincipalPermissionAttribute` class.

You can also combine the `Authenticated`, `Name`, and `Role` properties in a single attribute declaration. Doing so requires that the caller have all security permissions outlined. For example, to require that the user have the name Scott and along to the role Admin, use the following:

```
// C#
[PrincipalPermission(SecurityAction.Demand, Name="Scott", Role="Admin")]
public void SomeMethod(...)
{
   ...
}


' VB
<PrincipalPermission(SecurityAction.Demand, Name:="Scott", Role:="Admin")> _
Public Sub SomeMethod(...)
   ...
End Sub
```

To allow access for a caller that satisfies one or more conditions use multiple `PrincipalPermissionAttribute`s. For instance, the following declarations grant access to the method if the caller is user Scott or is in the role Admin:

```
// C#
[PrincipalPermission(SecurityAction.Demand, Name="Scott")]
[PrincipalPermission(SecurityAction.Demand, Role="Admin")]
public void SomeMethod(...)
{
   ...
}


' VB
<PrincipalPermission(SecurityAction.Demand, Name:="Scott")> _
<PrincipalPermission(SecurityAction.Demand, Role:="Admin")> _
Public Sub SomeMethod(...)
   ...
End Sub
```

Declaratively Applying Authorization Rules to Methods

The code available for download at the end of this article includes a simple business logic layer

implemented as a single class named `UserAPI.vb` in the `App_Code` folder. This class has three dummy methods:

- `SomeAdminAction` - a method meant only to be called by users in the `Admin` role
- `SomeAuthenticatedAction` - a method meant only to be called by authenticated users
- `SomeActionForScottOrSam` - a method meant only to be called by users Scott or Sam

To help enforce these authorization rules it is good practice to add `PrincipalPermissionAttribute`s to each method. The following code implements these attributes and ensures that the methods can only be executed by those users or roles that have permission.

```
Imports System.Security.Permissions

Public Class UserAPI
   <PrincipalPermission(SecurityAction.Demand, Role:="Admin")> _
   Public Sub SomeAdminAction()
      ' ...
   End Sub

   <PrincipalPermission(SecurityAction.Demand, Authenticated:=True)> _
   Public Sub SomeAuthenticatedAction()
      ' ...
   End Sub

   <PrincipalPermission(SecurityAction.Demand, Name:="Scott")> _
   <PrincipalPermission(SecurityAction.Demand, Name:="Sam")> _
   Public Sub SomeActionForScottOrSam()
      ' ...
   End Sub

End Class
```

These attributes can also be added to methods and event handlers in the code-behind classes of ASP.NET web pages.

Declaratively Applying Authorization Rules to Classes

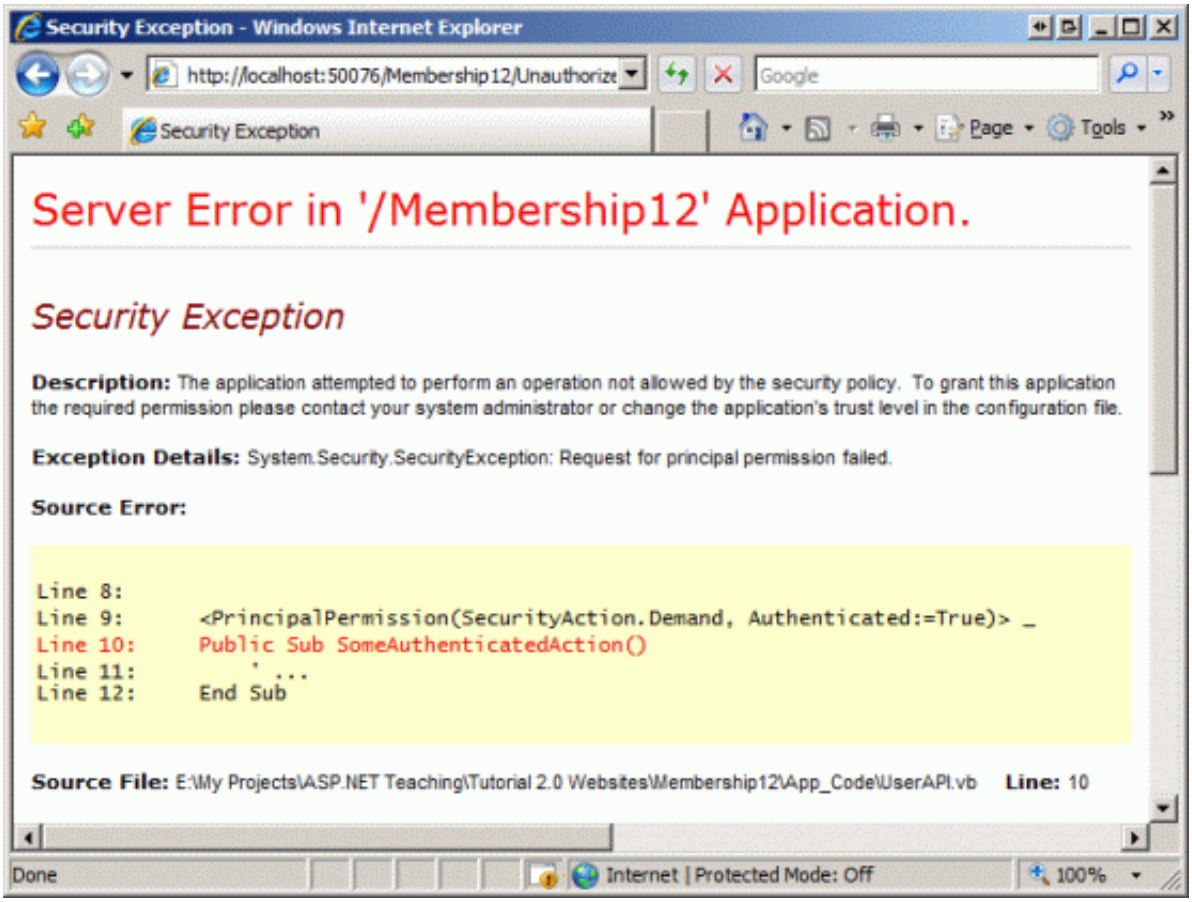
In addition to protecting particular methods, you can use `PrincipalPermissionAttribute` to protect an entire class. Simply add the attribute immediately above the class declaration. The demo for download at the end of this article uses this functionality to prohibit anonymous users from visiting the `WhoIsOnline.aspx` page. This is accomplished by adding a `PrincipalPermissionAttribute` attribute to the class declaration in the page's code-behind file:

```
Imports System.Security.Permissions

<PrincipalPermission(SecurityAction.Demand, Authenticated:=True)> _
Partial Class WhoIsOnline
   Inherits BasePage

   ...
End Class
```

What Happens When a Declarative Security Violation Occurs?

If an unauthorized caller executes a method or evokes a class that is decorated with a `PrincipalPermissionAttribute` attribute, a `SecurityException` exception is thrown. If there is no error handling the exception will bubble up to the ASP.NET runtime and will be handled there. Depending on how your website is configured this may display the "yellow screen of death" (see the following

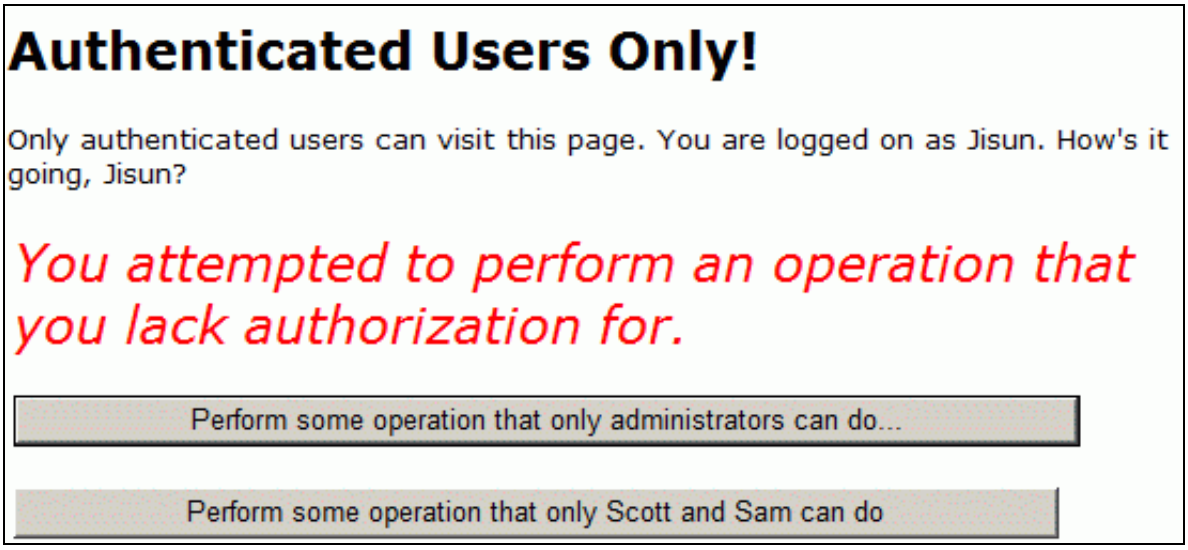
screenshot) or a custom error page.



This exception can be gracefully handled in a number of ways. You can surround the call to the protected methods and classes with a `Try...Catch` block, performing some action in the event of a `SecurityException`. The following code snippet and screenshot are from the code available for download at the end of this article, and show a simple `Try...Catch` block implementation. Specifically, if a `SecurityException` is raised when calling `SomeActionForScottOrSam` a message is displayed in a Label control (`Message`).

```
Try
   Dim api As New UserAPI()
   api.SomeActionForScottOrSam()
Catch sException As System.Security.SecurityException
   Message.Text = "You attempted to perform an operation that you lack authorization
for."
Catch ex As Exception
   Throw ex
End Try
```

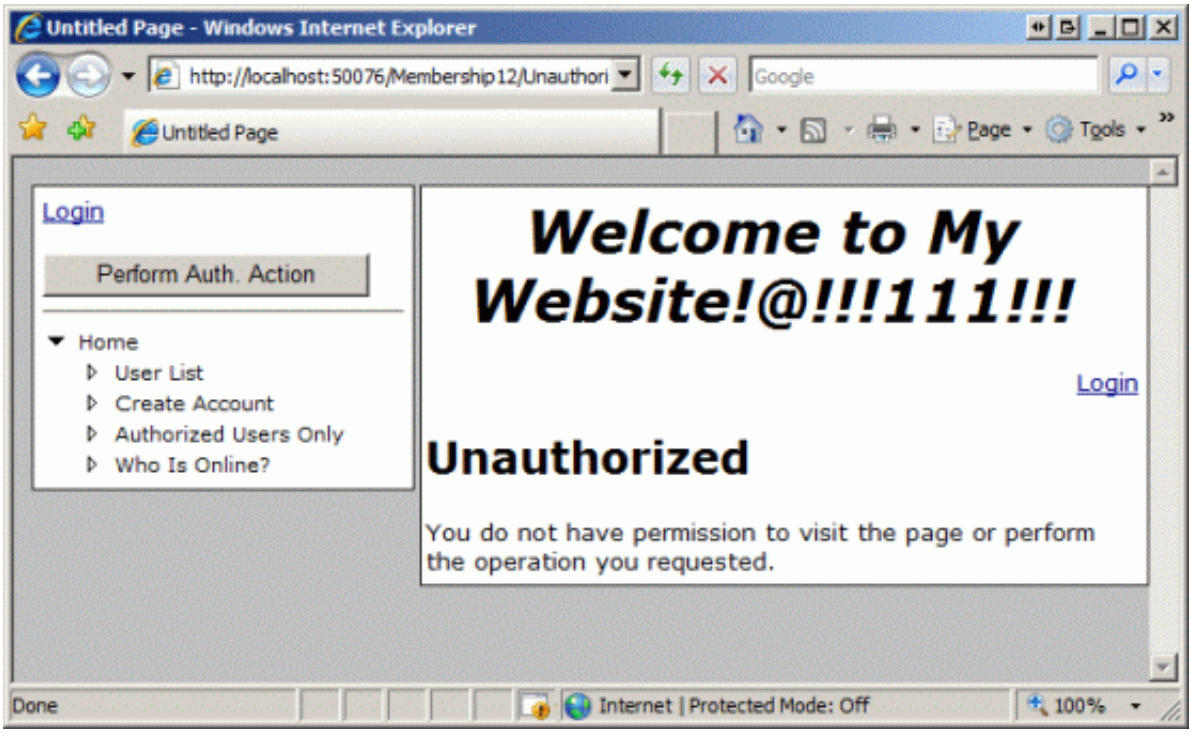
the website available for download from the end of this article has three user accounts: Scott, Jisun, and Sam. If Jisun logs on to the site, visits the page `~/UsersOnly/Default.aspx`, and clicks the button "Perform some operation that only Scott and Sam can do" the above code is executed. The `SomeActionForScottOrSam` method includes two `PrincipalPermissionAttribute` attributes that permit only users Scott and Sam. Consequently, when Jisun calls this method a `SecurityException` is thrown. The appropriate `Catch` block executes, displaying the message shown in the screenshot below.



Another option is to automatically redirect users to a particular web page in the event of a security exception. This can be accomplished by creating an event handler for the application's `Error` event in `Global.asax`. The `Error` event is raised whenever an unhandled exception bubbles up to the ASP.NET runtime. Here we can determine what type of exception was raised and take action based on that knowledge. The following event handler redirects the user to a page named `Unauthorized.aspx` whenever an unhandled `SecurityException` percolates up to the ASP.NET runtime.

```
Sub Application_Error(ByVal sender As Object, ByVal e As EventArgs)
   Dim err As Exception = Server.GetLastError()
   If TypeOf err Is System.Security.SecurityException Then
      Response.Redirect("~/Unauthorized.aspx?page=" & Server.UrlEncode(Request.RawUrl))
   End If
End Sub
```

With this event handler in place in the event of an unauthorized access the caller is automatically shown the following screen.



Use the `PrincipalPermissionAttribute` Attribute as a Last Resort

Keep in mind that the purpose of the `PrincipalPermissionAttribute` attribute is to serve as a final authorization check. You should still use your URL-based authorization rules in `web.config` along with user interface elements, such as the Login and LoginView controls. The `PrincipalPermissionAttribute` attribute is designed to protect your business logic from unauthorized callers should there be a security hole or oversight in the presentation layer.

If you download the demo at the end of this article you will no doubt notice the lack of preemptive checks in the presentation layer. For instance, there are buttons to perform actions that are only available to authenticated users, yet these buttons are displayed to both authenticated and anonymous users. Likewise, there are buttons to perform actions are only available to users in the Admin role, yet these buttons are shown to all, regardless of their role. This design was done purposefully so as to illustrate the behavior of the `PrincipalPermissionAttribute` attributes when an unauthorized user is encountered.

Conclusion

A well-designed website automatically prohibits users from visiting pages they are not authorized to see and hides user interface elements that perform actions that the currently logged on user cannot perform. However, it is easy to overlook a particular user interface element and accidentally show it to all users even though it performs an action that's reserved for a few. It's good practice to decorate your methods and classes with appropriate `PrincipalPermissionAttribute` attributes. These attributes provide a declarative means for defining what callers can access what classes and methods, and serves as a final check in case there are security holes or oversights in the presentation layer.

Happy Programming!

- By Scott Mitchell

Further Reading

- Forms Authentication, Authorization, Membership, and Roles Tutorials (includes VB & C# versions!)
- Tip/Trick: Adding Authorization Rules to Business and Data Layers using PrincipalPermissionAttributes
- Security Practices: ASP.NET 2.0 Security Practices at a Glance
- User-Based Authorization (VB version) | (C# version)
- Gracefully Responding to Unhandled Exceptions

Attachments

- Download the code used in this article

Article Information	
Article Title:	ASP.NET.Examining ASP.NET's Membership, Roles, and Profile - Part 12
Article Author:	Scott Mitchell
Published Date:	July 23, 2008
Article URL:	http://www.4GuysFromRolla.com/articles/072308-1.aspx